Unsupervised Deep Bug Report Summarization

Xiaochen Li  
Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province; School of Software, Dalian University of Technology, Dalian, China  
li1989@mail.dlut.edu.cn

He Jiang  
School of Software, Dalian University of Technology, Dalian, China; Beijing Institute of Technology  
jianghe@dlut.edu.cn

Dong Liu  
School of Software, Dalian University of Technology, Dalian, China  
dongliu@mail.dlut.edu.cn

Zhilei Ren  
School of Software, Dalian University of Technology, Dalian, China  
zren@dlut.edu.cn

Ge Li  
Key Laboratory of High Confidence Software Technologies (Peking University) Ministry of Education, China; Software Institute, Peking University, China  
lige@pku.edu.cn

ABSTRACT

Bug report summarization is an effective way to reduce the considerable time in wading through numerous bug reports. Although some supervised and unsupervised algorithms have been proposed for this task, their performance is still limited, due to the particular characteristics of bug reports, including the evaluation behaviours in bug reports, the diverse sentences in software language and natural language, and the domain-specific predefined fields. In this study, we conduct the first exploration of the deep learning network on bug report summarization. Our approach, called DeepSum, is a novel stepped auto-encoder network with evaluation enhancement and predefined fields enhancement modules, which successfully integrates the bug report characteristics into a deep neural network. DeepSum is unsupervised. It significantly reduces the efforts on labeling huge training sets. Extensive experiments show that DeepSum outperforms the comparative algorithms by up to 13.2% and 9.2% in terms of F-score and Rouge-n metrics respectively over the public datasets, and achieves the state-of-the-art performance. Our work shows promising prospects for deep learning to summarize millions of bug reports.

KEYWORDS

Bug Report Summarization, Mining Software Repositories, Deep Learning, Unsupervised Learning

1 INTRODUCTION

Bug repositories accumulate considerable knowledge for software projects [18, 47], including the experience on bug resolutions, historical software bugs, etc. To date, a single bug repository, e.g., the Eclipse Bugzilla repository, has already collected over 485,000 historical bug reports. Bug reports are important resources to maintain the long-term running of a software system, since the stakeholders of a software project prefer to understand the knowledge in these bug reports before conducting a software activity [33, 49]. For example, the common practice for software developers to fix newly reported bugs is to refer to similar historical bug reports for possible solutions [49]. Hence, nearly 600 sentences have to be read on average if a developer refers to only 10 historical bug reports [29]. Besides, bug reporters are usually required to wade through related bug reports before submitting a new one, to avoid a duplicate bug report submitted [33]. To reduce the tedious and time-consuming efforts in pursuing historical bug reports, bug report summarization is proven to be a promising direction [38]. The Debian community even encourages stakeholders to manually set a summary for each bug report [7], though the considerable human costs may burden this activity. Hence, automatic extractive bug report summarization is an alternative way, which aims to extract salient sentences in a bug report. Previous studies try to...
train bug report summarizers with features from conversation-based text summarization [17, 38]. Due to the limitation of labeled data, unsupervised algorithms are also migrating for this task [27, 29]. However, their performance is still limited.

Compared with texts of news, biographies, etc., bug reports are conversation-based. Salient and duplicate sentences mix together due to the frequent evaluation or assessment behaviours [27]. Second, bug reports contain many different sentence types in natural language and software language [29]. It is non-trivial to automatically measure the contributions of different sentence types to the summary. Third, a bug report is usually associated with some predefined fields [52], e.g., the component or the product causing this bug, which may provide helpful information in identifying salient sentences.

Based on the above characteristics, we propose a novel Deep learning based Summarizer, DeepSum, to better summarize bug reports. The kernel of DeepSum is a stepped auto-encoder network, which infers the bug report summary based on the hidden layers of the network. The basic idea of DeepSum is that, the hidden layers of a deep neural network act as an evaluation enhancement module. Then, it stepwise feeds the vectors of different sentence types into an auto-encoder network to automatically measure the weights of words in distinct sentence types to the summary. The words in the predefined fields are enhanced when initializing the network parameters. At last, DeepSum summarizes the bug report based on the word weights by Dynamic Programming.

We compare DeepSum against seven previous summarization techniques over all the public datasets for bug report summarization [17, 27, 29, 38]. Extensive experiments show that DeepSum significantly outperforms the comparative algorithms by up to 13.2% and 9.2% in terms of F-score and Rouge-n metrics respectively over the public datasets. Most parameters of DeepSum can be set in a wide range of values. Meanwhile, the unsupervised nature of DeepSum makes it independent to the datasets and costs little time to label huge datasets for training.

In summary, we make the following contributions.

(1) To the best of our knowledge, it is the first attempt towards employing deep learning to summarize bug reports.

(2) We propose a novel deep neural network in DeepSum for bug report summarization.

(3) Experiments demonstrate that DeepSum achieves the state-of-the-art performance for bug report summarization.

2 MOTIVATION

Bug reports are widely used for recording software bugs. Typically, two roles are involved in writing a bug report, including a reporter (mostly a user or a tester) to submit and discuss the details of the bug, and the participants who are the developers interested in fixing the bug. Generally, a bug report consists of a title, a description, some predefined fields and several comments. Fig. 1 is a bug report example [2]. The title of this report concludes the report topic, namely the slow speed of converting images. The details for reproducing the bug are added in the description. The predefined fields show this bug happened in the “General” component of the “GIMP” product. To find satisfactory solutions, several comments are added to the report by the reporter and participants.

Facing numerous lengthy bug reports, bug report summarization aims to generate an summary by directly extracting and highlighting informative or salient sentences (also called summary sentences) from the description and comments of a bug report [38] (the bold sentences in Fig. 1). Although several supervised and unsupervised algorithms have been proposed to resolve the problem, the unique characteristics of bug reports may weaken the effectiveness of these techniques.

First, bug reports are conversation-based text with frequent evaluation behaviours [27]. The reporter and the participants discuss other’s opinion by copying his/her sentence and adding evaluations to it. For example, in Fig. 1, before evaluating the sentence $s_8$, Xuân Baldauf wrote a similar sentence $s_9$ for evaluation. Another evaluation is that Adam D. Moss clicked the “reply” button to copy sentence $s_{10}$ as $s_{11}$ and $s_{12}$ (sentences start with “>”), and then evaluated $s_{10}$ in $s_{13}$. We call the sentences being evaluated as evaluated
sentences, e.g., 88, 810, and their similar versions as duplicate sentences, e.g., 89, 811, 812. Although the evaluated sentences are frequently discussed and important, their corresponding duplicate sentences make it hard to discover the salient one.

Second, bug reports consist of different sentence types, namely the natural language sentences by the reporter, the natural language sentences by the participants, and the software language sentences (typically, code snippets and system messages). It is crucial to measure the contributions of different sentence types. Specifically, the natural language sentences by the reporter are usually more informative than that by the participants [27], since participants’ comments are based on the topics proposed by the reporter. Meanwhile, despite developers are being familiar with the software language, a line in software language is usually less informative than that in natural language [29], e.g., a simple requirement “open a file” may result in many lines of code [42].

Third, a bug report is associated with many predefined fields [47, 52]. For these fields, product, component, version, and hardware are set by the reporter to reveal the environment information for reproducing the bug. Such information may be helpful for extracting salient sentences of a bug report.

Considering these unique characteristics, we propose DeepSum for effective bug reports summarization, a stepped auto-encoder network with evaluation enhancement and predefined fields enhancement modules to address these characteristics.

3 FRAMEWORK OF DEEPSUM
DeepSum summarizes a bug report by three steps, including bug report pre-processing, unsupervised network training, and summary generation. Given a new bug report, DeepSum first automatically detects a set of similar reports to form an unlabeled training set (Fig. 2(A)). The bug reports in the training set are used to train a stepped auto-encoder network for assigning sentence scores of the new bug reports (Fig. 2(B)). At last, DeepSum extracts summary sentences with Dynamic Programming based on these scores (Fig. 2(C)).

3.1 Bug Report Pre-processing
This step removes the noise in bug reports and collects similar reports for training the stepped auto-encoder network.

Bug reports are real-world data with considerable noises [49]. To remove the noises, DeepSum first tokenizes each sentence in the bug reports with a software-specific regular expression “\[w-]+\(\[w-]+\)*” [27]. Next, stop words removal [9] and Porter stemming [36] are conducted. At last, DeepSum filters sentences with less than three words [10], since they may not convey a piece of complete information.

After noise removal, DeepSum collects a set of bug reports for training the stepped auto-encoder network. For a new bug report (the bug report for summarizing), DeepSum first adds this new report to the training set. Since similar bug reports may contain common salient sentences or words to discuss bug solutions [4, 18], DeepSum also adds the top $k_1 - 1$ most similar reports prior to the new one from the same bug repository. The similarity is the cosine similarity between term frequency vectors of texts in description and comment fields of two bug reports calculated by Lucene [28]. We choose a simple and efficient way for calculation without consideration of the weight of each field or bug report topics.

At last, there are totally $k_1$ bug reports fed into the network. In this study, the size of the input training set $k_1$ is 100.

Example: Fig. 3(A) is an example representation of the new bug report in Fig. 1 and its top $k_1 - 1$ similar reports.

3.2 Unsupervised Network Training
This step trains a stepped auto-encoder network for summarization. We first transform the sentences in each training report into term frequency vectors and utilize evaluation enhancement to re-initialize the vectors. Then the vectors of different sentence types are fed into the network for training. In the network, words in predefined fields are enhanced during initializing the network parameters.

3.2.1 Evaluation Enhancement. Since there are frequent evaluation behaviours in bug reports, this sub-step enhances
the evaluated sentences and reduces the influence of the duplicate
for each training report. We represent a sentence
s in a bug report as \( f_s = \{f_1, f_2, \ldots, f_{\text{Voc}}\} \), where
\( f_{\text{Voc}} \) is the term frequency of the word \( f_{\text{Voc}} \) in the sentence
s normalized by the length of the bug report and \( \text{Voc} \) is the
vocabulary of the training set. Under this representation, the
similarity of two sentences is defined as the cosine similarity:

\[
sim(f_s, f_{s'}) = \frac{f_s \cdot f_{s'}}{|f_s||f_{s'}|},
\]

where \( f_s \) and \( f_{s'} \) are the term frequency vectors of the \( i \)th
sentence \( s_i \) and the \( j \)th sentence \( s_j \) respectively.

Based on the above definition, the evaluated and duplicate
sentences can be identified and re-initialized as follows.

(a) Identify a duplicate sentence. As shown in Fig. 1, a user
usually clicks the reply button or writes similar sentences
(e.g., the sentence \( s_9 \)) to evaluate a previous sentence. When
clicking the reply button, the Bugzilla system automatically
copies and adds “>” to the sentences (e.g., the sentence \( s_{11} \)).
DeepSum traverses sentences in a bug report from the last
sentence to the first one. A duplicate sentence is detected, if
the sentence starts with the sign “>” or its similarity with a
previous sentence exceeds a threshold. The threshold \( \theta \) is 0.9
as we will evaluate later in RQ1. These rules can be easily
migrated after observing other bug repositories. Meanwhile,
Bugzilla has been widely used by more than 136 companies
and organizations to manage bug reports [24].

(b) Identify the related evaluated sentence. DeepSum i-
extends the previous sentence with the largest similarity of
the duplicate one as the evaluated sentence. If two previous
sentences have the same similarity, DeepSum identifies
the sentence in front as the evaluated sentence.

c) Re-initialize the two vectors. DeepSum adds the value of
each element in the duplicate sentence vector to the evaluated
sentence vector, and then sets all the elements in the duplicate
sentence vector to be zero to reduce its influence.

The above process repeats until all the evaluated and
duplicate sentences are identified. As a result, the elements
in the evaluated sentence vector have larger initial values
than those in the duplicate sentence vectors.

Example: Fig. 3(A) shows the initial sentence vectors
of a new bug report. The vectors’ length is the vocabulary
of the training set. In Fig. 3(A), the first element in vector
\( s_8 \) is three, which means the first word in the vocabulary
occurs three times in \( s_8 \). For simplicity, we do not normalize
these elements in the example. After evaluation enhancement,
DeepSum identifies \( s_9 \) as a duplication of \( s_8 \), and regards
\( s_{11} \) and \( s_{12} \) as duplications of \( s_{10} \). Hence, the vector of \( s_9 \)
is added to \( s_8 \), e.g., the first element in vector \( s_8 \) is re-initialized
to 6 (in Fig. 3(B)). Similarly, \( s_{10} \) is re-set with \( s_{11} \) and \( s_{12} \).
Then, the elements in vectors \( s_9 \), \( s_{11} \) and \( s_{12} \) are set as zero.

3.2.2 Stepped Auto-encoder Network Training. With the
enhanced vectors of each training bug report, DeepSum trains
a stepped auto-encoder network for summarization.

As analyzed in Section 2, bug reports usually consist of
two or three types of sentences which may have distinct importance
to the summary, including the software language sentences
(referred as software sentences), the natural language sentences
by participators (participant sentences), and the
natural language sentences by the reporter (reporter sentences).
DeepSum detects the sentence types as follows.

(a) Detect the software sentences. Software sentences are
detected with the bug report analysis framework Infozilla [3].
For code snippets, it first identifies code lines by matching
some strong regular expressions, e.g., “=.*?;”, and then
expands the regions with surrounding sentences by many
weak rules, e.g., sentences contain “class”, “public”, etc. The
system messages are identified as the continuous sentences
containing “:” or “.”. Besides, if the distance of two regions
is within three sentences, we also take the sentences between
them as software language, in order to include any exceptions
to the above rules. We find that 95% detected software
sentences are true positive in our experiment.

(b) Detect the reporter sentences and participator sen-
tences. DeepSum directly classifies the remaining natural
language sentences by matching the reporter’s name with the
username item in the description and comment fields.

After detecting the sentence types, DeepSum stepwise en-
codes and decodes different sentence types with a stepped
auto-encoder network. In the following paragraphs, we intro-
duce the inputs and outputs, the architecture, and the
training process of the network.

(a) The network inputs and outputs. The network inputs
are three vectors. For a training bug report, DeepSum adds
up the sentence vectors of the same sentence types to form
three input vectors. These vectors denote as the software
vector \( f_i \), the participator vector \( f_2 \), and the reporter vector
\( f_3 \). After encoding and decoding the inputs with the network,
DeepSum requires the outputs of the network to be three vectors of the same length with the inputs, namely $f_1, f_2, f_3$, which means the reconstruction of the input vectors. The objective of the network is to minimize the differences between the input vectors and the output vectors.

(b) The network architecture. As shown in Fig. 2(B), there are five hidden layers in the network. The unit number of each hidden layer is similar with a traditional auto-encoder network [26], namely $num_{layer_1} = num_{layer_2} = 1000, num_{layer_3} = num_{layer_4} = 250$, and $num_{layer_5} = 10$. For this architecture, we have the following observations:

First, the network compresses three input vectors into 10 hidden units in layer_5, and then reconstructs the entire input vectors to the outputs. Hence, this hidden layer can be viewed as a compressed expression of the input vectors, which meets the concept of summarization. The compressed expression may reserve the meaningful information of the inputs to infer informative words of a new bug report.

Second, the network stepwise feeds the three inputs into the hidden layers. The software vector is at the bottom and the reporter vector is near the hidden layer layer_3. The reason is that, since software sentences are usually less informative than the other sentence types, DeepSum heavily compresses the software vector with three hidden layers, i.e., layer_1 to layer_3, to filter the noises. In contrast, the reporter sentences are usually more informative, DeepSum only compresses the reporter vector once to reserve all meaningful information.

(c) The training process. DeepSum inputs each training bug report to the network. By minimizing the differences between the inputs and outputs, DeepSum optimizes the network parameters with the widely used RMSProp optimizer [13]. The initial learning rate $\eta$ of the optimizer is 0.01 [19]. We also apply the dropout strategy [40] to prevent the network from overfitting. The dropout rate is 0.5 [40].

In this sub-step, DeepSum first encodes the inputs with layers layer_1 to layer_3. Let $f_1, f_2, f_3$ be the software vector, participator vector, and reporter vector respectively, $O_i$ be the output vector of layer_i (1 ≤ i ≤ 3), and $b_i$ be the bias of layer_i.

The matrices $U_i$ and $V_i$ are the weighs of connections between the fully-connected layer_i−1 or $f_1$ and layer_i. We have

$$O_i = \text{sig}(O_{i-1}U_{i-1}+f_iV_i+b_i),$$

where 1 ≤ i ≤ 3, $O_0 = 0$, $U_0 = 0$, and $\text{sig}(\cdot)$ is the sigmoid activation function. Then, DeepSum decodes the hidden layer layer_3 with layer_4 and layer_5. Let $f_4^*$, $f_5^*$ be the outputs corresponding to $f_1, f_2, f_3$. $O_i$ (4 ≤ i ≤ 5) represents the reconstruction of hidden layer layer_1 and layer_2, and $b_i$ and $c_j$ are the bias of $O_i$ and $f_j^*$ respectively. We have

$$O_i = \text{sig}(O_{i-1}U_{i-1}+b_i),$$

$$f_j^* = \text{sig}(O_{6-j}V_j^T+c_j),$$

where $U_{i-1} = U_{i-1}^T, 1 ≤ i ≤ 3$.

DeepSum optimizes the network parameters by minimizing the cross-entropy loss function [46] between $f = [f_1, f_2, f_3]$ and $f^* = [f_4^*, f_5^*, f_6^*]$. The parameters continue optimizing until the minimum loss retains unchanged within 100 iterations.

Example: In Fig. 3(C), DeepSum merges the vectors of the same sentence type. For the vectors in Fig. 2(B), it merges $s_6$ and $s_7$ into the software vector, merges $s_8, s_{11}$, and $s_{12}$ into the participator vector, and merges $s_1, s_9$, and $s_{10}$ into the reporter vector. Then, DeepSum feeds the three vectors of each training report into the network to train $U$ and $V$.

3.2.3 Predefined Fields Enhancement. Some predefined fields set by the reporter may be helpful for revealing a bug, including product, component, version, and hardware. DeepSum strengthens the influence of words in these fields in the same way as [26] when initializing the network parameters. Initially, the matrixes $U$, $V$ are randomly initialized to a zero mean Gaussian with a standard deviation of 0.01 [26]. If the $i$th word in the input vectors exists in the predefined fields, the parameters in row $i$ of $V$ are set to the maximum value in $V$. The reason is that, DeepSum assigns word weights according to the values of $U$ and $V$ (discussed in Section 3.3). The weights of the words in the predefined fields may be enhanced after maximizing the initial values of $U$ and $V$.

3.3 Summary Generation

This step assigns weights for words in the new bug report and collects a set of salient words. The salient words are used to assign scores for the bug report sentences. With the sentence scores, a summary is generated by Dynamic Programming.

3.3.1 Word Weighting. DeepSum assigns word weights according to the input vectors of the new bug report and the parameters $U$ and $V$. The intuitive idea is that, we assume the hidden layer (layer_3 in this study) is a compressed expression or summary of the input vectors. Hence, we can assign word weights by measuring how each word in the input vectors contributes to this compressed expression.

Specifically, we explain this process by an example of assigning the weight of word $i$ in the software vector (in Fig. 3(D)). We first collect the trained network parameters $U$ and $V$, and the input vectors of the new bug report. Then we multiply $V_1, U_1$ and $U_2$ to generate a new matrix $X_1$. For meaningful multiplication, a sigmoid function is applied on the matrices to transform the matrix values to be positive. The row number of $X_1$ is the same as the software vector length, and the column number is the unit number of layer_3, i.e., 10. $X_1$ is regarded as a simplified function to transform the software vector to layer_3. In this study, we utilize $U$ and $V$ to calculate the transformation function, since they are the core parameters of the network.

Then we assign the weight of word $i$. We set all the values in the software vector to be zero except word $i$. In Fig. 3(D), the first element "1" in the vector is reserved. We can multiply this vector with $X_1$ to get the importance of word $i$ to each of the ten hidden units in layer_3. We average the ten values to achieve the weight of word $i$ in software sentences. Similarly, the weights of word $i$ in participator sentences and reporter sentences can be calculated by $V_2$ and $U_2$, or $V_3$. DeepSum sums the three weights of word $i$, and ranks and selects the top $k_2$ words in the new bug report as a salient words set.
3.3.2 Sentence Weighting. Based on the salient words set, we assign scores for the sentences in the new bug report. For each sentence, we collect its enhanced sentence vector, and multiple the elements in the vector with the summed weights of the corresponding word in the salient words set.

Example: Fig. 3(E) lists the five salient words (k₅=5), their indexes, and the summed word weights in the example. Words “black” and “image” are the first two element in the vocabulary (indexed as 1 and 2), and the word “use” is the last one (indexed as 4773). In the example, the sentence score of s₁₀ is achieved by a multiplication between the enhanced term frequencies in the sentence vector of s₁₀ (in Fig. 3(B)) and the summed word weights in the salient words set, namely s₁₀ is scored as 0*0.03+6*0.02+...+5*0.03. All the weights of words excluded in the salient words set are zero.

3.3.3 Sentence Selection. With the sentence scores, bug report summarization turns into selecting sentences sₘₐₓ ∈ sₜₒₐₚₑₗ to maximize the total sentence score under a length limitation. It is a typical 0-1 Knapsack problem, which can be solved by Dynamic Programming [43], a common sentence selection algorithm in text summarization [26, 31]. Similar to previous studies [27, 38], we select sentences until they reach 25% length of the new bug report in words. The idea of Dynamic Programming is to decide whether to add sₙ to the summary under the remaining summary length limitation, when the maximum total sentence score of sₙ to sₙ₋₁ is already achieved.

4 EXPERIMENT SETUP

DeepSum is implemented with the machine intelligence library Tensorflow [1]. It runs on Ubuntu 16.04 with Intel Core(TM) i7-6700 CPU, GTX1080 GPU and 16G memory.

4.1 Dataset

We evaluate the summarizers over two public bug report datasets, namely SDS (Summary DataSet) [38] and ADS (Authorship DataSet) [17], consisting of 36 and 96 bug reports respectively. Each bug report in the datasets is annotated by three annotators. The annotators are asked to conclude the report in around 25% length of the report in their own words. The concluded sentences are called an Abstractive Reference summary AbsRef. Then the annotators link each sentence in AbsRef to one or more sentences in the original bug report. The Extractive Reference summary ExtRef consists of the sentences linked by at least two annotators.

4.2 Baseline Algorithms

We compare DeepSum with seven algorithms in previous studies. These algorithms summarize bug reports from multiple aspects, including taking advantage of the conversation-based characteristic of bug reports [BRC[38], ACS[17]], transferring classical text summarizers to bug reports (Centroid, MMR, Grasshopper, DivRank) [29], and manually mining features from bug reports for summarization (Hurried [27]).

We reproduce the baseline algorithms, since previous studies evaluate their algorithms under different criteria [29, 38]. In the experiments, an algorithm selects summary sentences until they reach 25% length of the new bug report in words [17, 27, 38]. For the supervised algorithms BRC and ACS, we conduct the Leave-One-Out (LOO) framework [16, 38] and Two-Fold Cross-Validation (TFCV) framework [41] for evaluation, denoted as BRC_LOO and BRC_TFCV, and ACS_LOO and ACS_TFCV. LOO framework is used by all the reports in the same dataset except for one for summarization. TFCV framework [41] randomly splits the dataset into two parts for training and testing. LOO framework requires more labeled training data. Due to the lack of public labeled datasets for bug report summarization, TFCV framework may better reflect the actual performance of the supervised algorithms.

4.3 Evaluation Metrics

We first evaluate the algorithms with four metrics proposed in previous studies for bug report summarization [38], namely Precision, Recall, F-score, and Pyramid score. For a set of selected summary sentences sₘₐₓ ∈ sₜₒₐₚₑₗ, these metrics are:

\[ \text{Precision} = \frac{\text{Num}_{\text{hit}}}{\text{Num}_{\text{selected}}} \]
\[ \text{Recall} = \frac{\text{Num}_{\text{hit}}}{\text{Num}_{\text{ExtRef}}} \]
\[ \text{F-score} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \]
\[ \text{Pyramid} = \frac{\text{Num}_{\text{TotalLinks}}}{\text{Num}_{\text{MaxLinks}}} \]

where \( \text{Num}_{\text{selected}} \) is the number of sentences in sₘₐₓ, \( \text{Num}_{\text{hit}} \) is the number of sentences in sₘₐₓ which belongs to ExtRef, \( \text{Num}_{\text{ExtRef}} \) is the number of sentences in ExtRef, \( \text{Num}_{\text{TotalLinks}} \) is the total number of times that sentences in sₘₐₓ are linked by the annotators, while \( \text{Num}_{\text{MaxLinks}} \) is the maximum possible links for the same number of sentences.

We further evaluate the algorithms with the widely accepted summarization evaluation package Rouge-1.5.5 [23], which evaluates sₘₐₓ with the human written summaries AbsRef. In this study, the metrics\(^1\) R-1 and R-2 are used, due to their abilities in emulating human evaluation procedures [5, 34].

\[ \text{Rouge-n} = \frac{\sum_{s \in \text{AbsRef}} \sum_{\text{gram}_n \in s} \text{Count}_{\text{match}}(\text{gram}_n)}{\sum_{s \in \text{AbsRef}} \sum_{\text{gram}_n} \text{Count}(\text{gram}_n)} \]

where \( n \) is the n-gram length, \( \text{Count}_{\text{match}}(\text{gram}_n) \) is the number of n-gram co-occurring in both sₘₐₓ and AbsRef. For metrics R-1 and R-2, the values of \( n \) are 1 and 2 respectively.

4.4 Research Questions (RQ)

RQ1: Do the parameters influence DeepSum’s performance?
RQ2: How do the evaluation enhancement and predefined field enhancement modules influence DeepSum’s performance?
RQ3: Is DeepSum effective in assigning word weights compared to some alternative strategies?
RQ4: How does DeepSum perform against baselines?
RQ5: How does DeepSum perform under different sentence selection criteria?
5 EXPERIMENT RESULTS

5.1 Answer to RQ1: Parameter Influence

Fig. 4 shows the influence of similarity value \( \theta \), training set size \( k_1 \), and salient words number \( k_2 \) with respect to distinct evaluation metrics over the two datasets.

5.1.1 Similarity Value \( \theta \). DeepSum detects the duplicate versions of an evaluated sentence with \( \theta \). To tune \( \theta \), we fix \( k_1=100 \) and \( k_2=10 \), and vary \( \theta \) from 0 to 1 with a step size of 0.1. When \( \theta=1 \), no similar sentences are considered as duplications. When \( \theta=0 \), all the sentences are regarded as the duplication of the first sentence. Since the first sentence tends to be important in a bug report, when \( \theta \) is small (\( \theta < 0.3 \)), Precision is high. However, other metrics are low as all the other sentences are regarded as the duplications with zero element vectors. When \( \theta \) increases from 0.6 to 0.9, the metrics turn to be stable. We set \( \theta=0.9 \) in the following experiments.

5.1.2 Training Set Size \( k_1 \). DeepSum expands the new bug report with similar ones in the same repository. To tune \( k_1 \), we set \( \theta=0.9 \) and \( k_2=10 \). When \( k_1=1 \), it means only the new bug report for summarization is fed into the deep learning network which may overfit the network. When \( k_1 \) increases to 300, the performance of DeepSum begins to drop slightly, since many unrelated bug reports are included in the training set. DeepSum performs best when \( k_1 \) ranges from 100 to 150.

5.1.3 Salient Words Number \( k_2 \). The salient words number is used to assign sentence scores in the new bug report. To investigate \( k_2 \), we set \( \theta=0.9 \) and \( k_1=100 \), and \( k_2 \) varies from 1 to 15 with a step size of 1. When \( k_2=1 \), it means DeepSum weights the sentences only based on the word of the largest weight in the bug report. Although this word may repeatedly occur in the ExtRef summary, it loses the diversity of the summary, which leads to a relatively high Precision but low Recall. As \( k_2 \) increases, the performance is relatively stable, namely DeepSum is insensitive to \( k_2 \). According to this observation, we set \( k_2=10 \) in the experiments.

For \( k_2 \), we avoid to set a large value. First, we find as \( k_2 \) increases, long sentences with large amount of words tend to have high sentence scores, which makes DeepSum fail to select the right summary sentences. Second, when \( k_2=10 \), DeepSum can cover the majority of sentences in a bug report. We calculate the rate of covered sentences over distinct \( k_2 \) in Fig. 5. We count the number of sentences that sentence scores are greater than zero and divided this number by the number of sentences in a dataset. When \( k_2=1 \), DeepSum can assign scores to around 20% sentences in the datasets, namely 80% sentence scores are zero. Thus, it is hard to rank and select the 80% sentences. When \( k_2=10 \), around 60% sentences are assigned by DeepSum. Since the remaining sentences may be filtered during pre-processing or not contain any informative words, DeepSum is able to select the correct summary sentences then.

Conclusion. By setting these parameters, DeepSum can handle diverse situations as different metrics are preferred, e.g., Precision or Recall. Most parameters of DeepSum can be set in a wide range of values.

5.2 Answer to RQ2: Module Influence

DeepSum utilizes a stepped auto-encoder network with evaluation enhancement module and predefined field enhancement module to take advantages of bug report characteristics. Table 1 shows the influence of these modules. The sign “A” denotes the evaluation enhancement module and “B” denotes the predefined fields enhancement module. The sign “\( \times \)” means running DeepSum without a certain module.

![Figure 5: Rate of covered sentences over distinct \( k_2 \)](image-url)
Table 1: Influence of Different Modules.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>Precision</th>
<th>Recall</th>
<th>F-score</th>
<th>Pyramid</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDS</td>
<td>√</td>
<td>√</td>
<td>0.621</td>
<td>0.388</td>
<td>0.462</td>
<td>0.624</td>
<td>0.503</td>
<td>0.177</td>
</tr>
<tr>
<td></td>
<td>×</td>
<td>√</td>
<td>0.533</td>
<td>0.397</td>
<td>0.397</td>
<td>0.551</td>
<td>0.543</td>
<td>0.160</td>
</tr>
<tr>
<td></td>
<td>√</td>
<td>×</td>
<td>0.600</td>
<td>0.381</td>
<td>0.450</td>
<td>0.598</td>
<td>0.552</td>
<td>0.166</td>
</tr>
<tr>
<td></td>
<td>×</td>
<td>×</td>
<td>0.529</td>
<td>0.339</td>
<td>0.399</td>
<td>0.544</td>
<td>0.540</td>
<td>0.160</td>
</tr>
<tr>
<td>ADS</td>
<td>√</td>
<td>√</td>
<td>0.606</td>
<td>0.394</td>
<td>0.457</td>
<td>0.681</td>
<td>0.553</td>
<td>0.249</td>
</tr>
<tr>
<td></td>
<td>×</td>
<td>√</td>
<td>0.586</td>
<td>0.382</td>
<td>0.443</td>
<td>0.661</td>
<td>0.548</td>
<td>0.243</td>
</tr>
<tr>
<td></td>
<td>√</td>
<td>×</td>
<td>0.577</td>
<td>0.371</td>
<td>0.432</td>
<td>0.661</td>
<td>0.548</td>
<td>0.246</td>
</tr>
<tr>
<td></td>
<td>×</td>
<td>×</td>
<td>0.558</td>
<td>0.359</td>
<td>0.418</td>
<td>0.639</td>
<td>0.543</td>
<td>0.242</td>
</tr>
</tbody>
</table>

Table 2: Performance on Word Weighting Strategies.

<table>
<thead>
<tr>
<th></th>
<th>Precision</th>
<th>Recall</th>
<th>F-score</th>
<th>Pyramid</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDS AE Strategy</td>
<td>0.581</td>
<td>0.370</td>
<td>0.437</td>
<td>0.590</td>
<td>0.555</td>
<td>0.165</td>
</tr>
<tr>
<td>DeepSum</td>
<td>0.621</td>
<td>0.388</td>
<td>0.462</td>
<td>0.624</td>
<td>0.568</td>
<td>0.177</td>
</tr>
<tr>
<td>ADS AE Strategy</td>
<td>0.577</td>
<td>0.366</td>
<td>0.427</td>
<td>0.657</td>
<td>0.549</td>
<td>0.240</td>
</tr>
<tr>
<td>DeepSum</td>
<td>0.606</td>
<td>0.394</td>
<td>0.457</td>
<td>0.681</td>
<td>0.553</td>
<td>0.249</td>
</tr>
</tbody>
</table>

Obviously each module has its own contribution to the final summary. The evaluation enhancement module identifies frequently discussed sentences in a bug report and filters the duplicate copies. The predefined fields enhancement module leverages the information in the predefined fields to identify the salient words. In addition, when we remove both modules, the performance is the worst which means these modules are both useful for summarizing bug reports.

We find that in SDS, removing the evaluation enhancement module leads to a sharp drop on most metrics, e.g., F-score drops from 0.462 to 0.397. In contrast, this module has less impact on ADS. The reason is that, the average number of sentences per bug report in SDS is 65. The evaluation behaviours happen frequently in these long bug reports. While, the average number of sentences is only 39 in ADS. Most bugs are fixed before the evaluation behaviours happen. However, the results show that the modules of DeepSum work well regardless of datasets with distinct bug reports length.

Conclusion. The evaluation enhancement module and predefined fields enhancement module have positive influence on DeepSum. These modules work well over different datasets.

5.3 Answer to RQ3: Word Weighting

This RQ investigates whether the word weighting strategy of DeepSum in Section 3.3.1 is useful. For this purpose, we replace the word weighting strategy with two alternative ones and keep the other modules of DeepSum.

TF Strategy. The first one is a Term Frequency based strategy. It investigates whether a deep neural network is necessary. For a new bug report, TF strategy first transforms the sentences in the report into vectors in the same way as DeepSum. For fair comparison, the evaluation enhancement module is applied on each sentence vector. Then we calculate and rank the words according to their term frequency. The top 10 words are selected as the salient words. The weights of words are the values of term frequency. Based on the selected words and enhanced sentence vectors, TF strategy assigns sentence scores in the same way as DeepSum.

AE Strategy. The second one is an Auto-Encoder based strategy. It investigates whether the stepped auto-encoder network outperforms a standard auto-encoder network in word weighting. Compared with DeepSum in Fig. 2(B), a standard auto-encoder only has one input layer $f_1$ and output layer $f_1^*$, i.e., we remove the inputs $E_2$, $E_3$ and the outputs $E_2^*$, $f_1^*$. AE strategy works as follows. After collecting the input training set, we merge all the sentence vectors of different sentence types into a single vector and feed it to $f_1$. AE strategy conducts the same training and sentence selection procedure as DeepSum. The evaluation enhancement and predefined field enhancement modules are also applied. Unlike DeepSum, AE strategy assigns weights of all words based on a three layer compression, namely assigning word weights based on $V_1$, $U_1$, $U_2$ and the single input vector.

Table 2 shows the performance of different word weighting strategies. DeepSum outperforms both alternative strategies. For example, DeepSum has a relative improvement by up to 5.7% and 7.5% in terms of F-score over the two datasets. The results show that the stepped auto-encoder network is superior to a simple word weighting strategy in assigning word weights, i.e., TF strategy. Meanwhile, words in different sentence types make distinct contributions to the summary. Compared with a standard auto-encoder network (AE strategy), DeepSum fully leverages the sentence type information to make a more accurate measurement on words.

Conclusion. DeepSum’s word weighting strategy outperforms the alternatives, i.e., TF strategy and AE strategy.

5.4 Answer to RQ4: Overall Performance

We compare DeepSum with previous algorithms for bug report summarization [17, 27, 29, 38]. Table 3 and Table 4 show the overall performance of DeepSum averaged by ten times’ running. A bold value is the best result among all the algorithms. A grey cell means DeepSum outperforms an algorithm with p<0.05 by the paired Wilcoxon signed rank test under Holm’s correction [14].

Overall, DeepSum outperforms the comparative algorithms in terms of the majority of the metrics. It achieves the state-of-the-art results on 4 and 6 metrics over the two datasets. On the SDS dataset, DeepSum outperforms the other algorithms by up to 0.119 and 0.092 in terms of F-score and R-1 respectively. On the ADS dataset, DeepSum still achieves equal or better performance than the comparative algorithms. Although different metrics evaluate an algorithm from their unique aspects, DeepSum outperforms these algorithms in most cases and is stable on distinct metrics and datasets.

We note that the training sets heavily affect the supervised algorithms. The ACS algorithm performs well on ADS in a Leave-One-Out framework, since ADS contains numerous labeled reports written by the same reporter which fits for ACS. However, neither such a training set or the training set for ACS. Hence, when it comes to SDS that contains few reports by the same reporter, ACS...
Table 3: Overall Comparison on SDS.

<table>
<thead>
<tr>
<th>Precision</th>
<th>Recall</th>
<th>F-score</th>
<th>Pyramid</th>
<th>R-1</th>
<th>R-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRC_LOO</td>
<td>0.570</td>
<td>0.350</td>
<td>0.400</td>
<td>0.630</td>
<td>0.521</td>
</tr>
<tr>
<td>BRC_FCV</td>
<td>0.524</td>
<td>0.321</td>
<td>0.362</td>
<td>0.580</td>
<td>0.493</td>
</tr>
<tr>
<td>ACS_LOO</td>
<td>0.595</td>
<td>0.337</td>
<td>0.400</td>
<td>0.604</td>
<td>0.516</td>
</tr>
<tr>
<td>ACS_FCV</td>
<td>0.562</td>
<td>0.310</td>
<td>0.359</td>
<td>0.572</td>
<td>0.488</td>
</tr>
<tr>
<td>Centroid</td>
<td>0.536</td>
<td>0.269</td>
<td>0.343</td>
<td>0.460</td>
<td>0.471</td>
</tr>
<tr>
<td>MMR</td>
<td>0.617</td>
<td>0.353</td>
<td>0.429</td>
<td>0.551</td>
<td>0.498</td>
</tr>
<tr>
<td>Grasshopper</td>
<td>0.525</td>
<td>0.300</td>
<td>0.368</td>
<td>0.521</td>
<td>0.505</td>
</tr>
<tr>
<td>DivRank</td>
<td>0.591</td>
<td>0.301</td>
<td>0.378</td>
<td>0.546</td>
<td>0.500</td>
</tr>
<tr>
<td>Hurried</td>
<td>0.710</td>
<td>0.300</td>
<td>0.410</td>
<td>0.710</td>
<td>0.525</td>
</tr>
<tr>
<td>DeepSum</td>
<td>0.621</td>
<td>0.388</td>
<td>0.462</td>
<td>0.624</td>
<td>0.553</td>
</tr>
</tbody>
</table>

Table 4: Overall Comparison on ADS.

<table>
<thead>
<tr>
<th>Precision</th>
<th>Recall</th>
<th>F-score</th>
<th>Pyramid</th>
<th>R-1</th>
<th>R-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRC_LOO</td>
<td>0.568</td>
<td>0.350</td>
<td>0.412</td>
<td>0.659</td>
<td>0.517</td>
</tr>
<tr>
<td>BRC_FCV</td>
<td>0.528</td>
<td>0.314</td>
<td>0.388</td>
<td>0.620</td>
<td>0.492</td>
</tr>
<tr>
<td>ACS_LOO</td>
<td>0.605</td>
<td>0.391</td>
<td>0.452</td>
<td>0.671</td>
<td>0.546</td>
</tr>
<tr>
<td>ACS_FCV</td>
<td>0.556</td>
<td>0.343</td>
<td>0.400</td>
<td>0.625</td>
<td>0.520</td>
</tr>
<tr>
<td>Centroid</td>
<td>0.488</td>
<td>0.280</td>
<td>0.337</td>
<td>0.561</td>
<td>0.473</td>
</tr>
<tr>
<td>MMR</td>
<td>0.505</td>
<td>0.356</td>
<td>0.395</td>
<td>0.585</td>
<td>0.503</td>
</tr>
<tr>
<td>Grasshopper</td>
<td>0.446</td>
<td>0.337</td>
<td>0.362</td>
<td>0.548</td>
<td>0.504</td>
</tr>
<tr>
<td>DivRank</td>
<td>0.445</td>
<td>0.282</td>
<td>0.325</td>
<td>0.545</td>
<td>0.498</td>
</tr>
<tr>
<td>Hurried</td>
<td>0.580</td>
<td>0.349</td>
<td>0.418</td>
<td>0.637</td>
<td>0.544</td>
</tr>
<tr>
<td>DeepSum</td>
<td>0.606</td>
<td>0.394</td>
<td>0.457</td>
<td>0.681</td>
<td>0.553</td>
</tr>
</tbody>
</table>

Table 5: Top Words for Bug Report #170801.

<table>
<thead>
<tr>
<th>Sentence type</th>
<th>Five salient words</th>
</tr>
</thead>
<tbody>
<tr>
<td>f₁: software sent.</td>
<td>palette colormap gimpimageconvert.c, palette_type option</td>
</tr>
<tr>
<td>f₂: participant sent.</td>
<td>palette image convert use difference</td>
</tr>
<tr>
<td>f₃: reporter sent.</td>
<td>black operation image 1bit tool</td>
</tr>
<tr>
<td>Final</td>
<td>black image 1bit tool use</td>
</tr>
</tbody>
</table>

Figure 6: Results on varied summary lengths.

However, the variation of Precision is small (<10%). In contrast, there is a more than 40% promotion on Recall. The Recall values reach 0.692 and 0.611 on SDS and ADS respectively when we select a summary of 70% bug report length. It implies that DeepSum successfully ranks the sentences in the ExtRef summary higher than the other sentences. Similarly, other metrics also increase. For example, F-score significantly increases to 0.532 for SDS and 0.512 for ADS at 40% length. Then, F-score is stable with less than 5% increment when the summary length further increases.

However, we note that there is no an optimal summary length in practise, due to the different reading habits of developers. Hence, after assigning the scores of sentences in a bug report, DeepSum is able to efficiently generate the summary of different lengths by Dynamic Programming.

Conclusion. DeepSum works well at different summary lengths. With a summary of 40% length, DeepSum can cover more than half of the sentences in the ground-truth summary.

6 DISCUSSION

6.1 Example by DeepSum

Taking the bug report in Fig. 1 as an example, Table 5 lists the top 5 words weighted by DeepSum with respect to different sentence types. The network summarizes bug reports in three aspects. For the input of software sentences, the top 5 words are variables (e.g., colormap, palette_type) or file names (e.g., gimpimage-convert.c) regarding the code snippets. The important words of the participator sentences focus on the solutions (e.g., use, different, palette) to the reported bug. While the salient words of the reporter sentences address the core problems of the report, namely conversing 1bit image between black and white is slow. DeepSum stepwise weights sentences of different types according to their contributions to the summary. As a result, most information of the reporter’s sentences is reserved (e.g., black, 1bit); some information of the participators’ solutions is supplemented (e.g., use);
and massive information of the software language is filtered. Since every sentence type may contribute to the summary, DeepSum automatically decides the filtered information in this process. It shows DeepSum’s ability in measuring words according to different sentence types of bug reports.

Table 6 is a partial summary for the bug report in Fig. 1 by DeepSum. In practice, these sentences are labeled in bold on the original bug report for developers to understand their context. The summary is in accordance with the above observations. Sentences #1-#3 are the phenomena and reproducing steps of the bug, while sentences #4-#6 are the possible solutions and some important discussions between the reporter and participants.

### 6.2 Threats to Validity

The generality of DeepSum should be further studied. To alleviate this threat, we evaluate DeepSum on all the public datasets for bug report summarization. We also compare DeepSum with seven previous summary algorithms over multiple metrics.

The running time of DeepSum is another threat in real developing scenarios [15]. DeepSum is a deep neural network based algorithm which takes 5.66 minutes on average to summarize a bug report, including 5.57 minutes in training the network. Since bug report summarization is usually used for perusing historical bug reports, most reports could be summarized in an off-line mode. Meanwhile, bug reports in several years ago may receive less attention due to the changes of the software architecture and source code. Additionally, computing in the cloud could also shorten the running time.

### 7 RELATED WORK

#### 7.1 Bug Report Summarization

Bug report summarization techniques can be classified into supervised and unsupervised ones. For supervised techniques, Rastkar et al. migrate features from email summarization to train bug report summarizers [38]. They find that labeling domain-specific data sets is important to improve the performance of bug report summarizers. Jiang et al. [17] summarize bug reports in consideration of the reporters’ authorship.

Due to the limitation of labeled data, many unsupervised text summarization and sentence selection strategies are also employed [29], including Centroid [37], Maximal Marginal Relevance (MMR) [6], Grasshopper [51], and Diverse Rank (DivRank) [30]. Lotufo et al. [27] propose a graph-based unsupervised algorithm by simulating human reading behaviors, which achieves relatively high Precision by sacrificing Recall. In this study, DeepSum is a novel unsupervised algorithm for bug report summarization, which assigns weights of words and sentences without human mined features.

#### 7.2 Deep Neural Networks for Software


In contrast to previous studies, we attempt to apply a novel deep neural networks to summarize bug reports, a complex software artifact with natural languages and source code.

### 8 CONCLUSION

Bug reports are crucial to fix software bugs. In this study, we propose an unsupervised deep learning algorithm for bug report summarization. Our model fully leverages the characteristics of bug reports. Experiments over two public bug report datasets show that our model outperforms the comparative algorithms significantly by adopting domain-specific characteristics. In the future, we plan to conduct case studies to investigate whether such an automated model could facilitate millions of software developers in perusing bug reports in a real developing scenario. More information about DeepSum is at http://oscar-lab.org/people/~xcli/open/deepsun/

### ACKNOWLEDGMENTS

This work is supported in part by the National Natural Science Foundation of China under Grants 61722202, 61772107.
REFERENCES


